**读书 先知其然，然后知其所以然，docker，先会用，再研究源码**

**docker技术入门与实战读书笔记(一)**

**声明:以下内容很多摘自<>这本书,主要记录一些笔记,方便查看**

**1. docker简介**

**1.1 什么是docker**

* docker基于Linux内核的cgroup,namespace及AUFS类的Union FS 等技术,对进程进行封装隔离,属于操作系统层面的虚拟化技术。由于隔离的进程独立于宿主和其它的隔离的进程,因此也称其为容器。Docker在容器的基础上,进行了进一步的封装,从文件系统、网络互联到进程隔离等等,极大的简化了容器的创建和维护。使得Docker技术比虚拟机技术更为轻便、快捷。
* 传统虚拟机技术是虚拟出一套硬件后,在其上运行一个完整操作系统,在该系统上再运行所需应用进程;而容器内的应用进程直接运行于宿主的内核,容器内没有自己的内核,而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。

### 1.2 为什么使用docker

* 更高效  
  由于容器不需要进行硬件虚拟以及运行完整操作系统等额外开销,Docker 对系统资源的利用率更高。无论是应用执行速度、内存损耗或者文件存储速度,都要比传统虚拟机技术更高效。
* 更快速  
  Docker由于直接运行于宿主内核,无需启动完整的操作系统,因此可以做到秒级、甚至毫秒级的启动时间。

## 2. 基本概念

### 2.1 docker镜像

Docker镜像(Image),就相当于是一个root文件系统。比如ubuntu:16.04就包含了完整的一套Ubuntu16.04最小系统的root文件系统。

Docker镜像是一个特殊的文件系统,除了提供容器运行时所需的程序、库、资源、配置等文件外,还包含了一些为运行时准备的一些配置参数(如匿名卷、环境变量、用户等)。镜像不包含任何动态数据,其内容在构建之后也不会被改变。

镜像并非是像一个ISO那样的打包文件,镜像只是一个虚拟的概念,其实际体现并非由一个文件组成,由多层文件系统联合组成。

镜像构建时,会一层层构建,前一层是后一层的基础。每一层构建完就不会再发生改变,后一层上的任何改变只发生在自己这一层。因此,在构建镜像的时候,需要额外小心,每一层尽量只包含该层需要添加的东西.

### 2.2 docker容器

镜像(Image)和容器(Container)的关系,就像是面向对象程序设计中的类和实例一样,镜像是静态的定义,容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。

容器的实质是进程,但与直接在宿主执行的进程不同,容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的root文件系统、自己的网络配置、自己的进程空间,甚至自己的用户ID空间。

每一个容器运行时,是以镜像为基础层,在其上创建一个当前容器的存储层,我们可以称这个为容器运行时读写而准备的存储层为容器存储层。

**容器存储层的生存周期和容器一样,容器消亡时,容器存储层也随之消亡。因此,任何保存于容器存储层的信息都会随容器删除而丢失。**

**按照Docker最佳实践的要求,容器不应该向其存储层内写入任何数据,容器存储层要保持无状态化。所有的文件写入操作,都应该使用数据卷(Volume)、或者绑定宿主目录,在这些位置的读写会跳过容器存储层,直接对宿主(或网络存储)发生读写,其性能和稳定性更高。**

### 2.3 仓库

镜像构建完成后,可以很容易的在当前宿主机上运行,但是,如果需要在其它服务器上使用这个镜像,就需要一个集中的存储、分发镜像的服务,**Docker Registry**就是这样的服务。  
一个**Docker Registry**中可以包含多个**仓库(Repository)**;每个仓库可以包含多个标签(Tag);每个标签对应一个镜像。

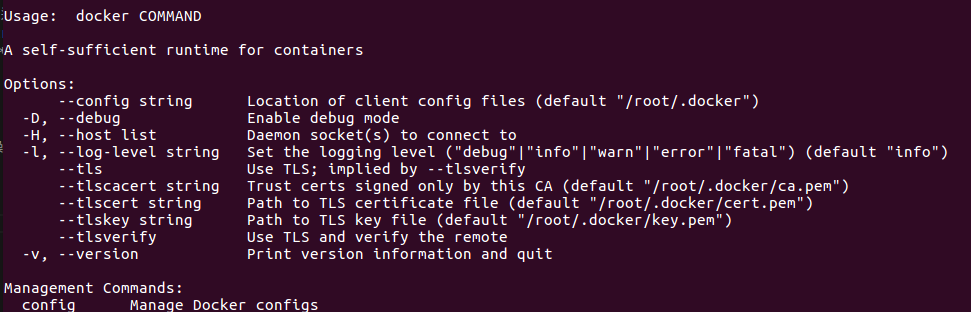
## 3. 镜像的简单使用

### 3.1 使用帮助

安装docker之后,可以使用docker命令进行操作,如果不了解如何使用,可以用

docker --help

结果如下所示:

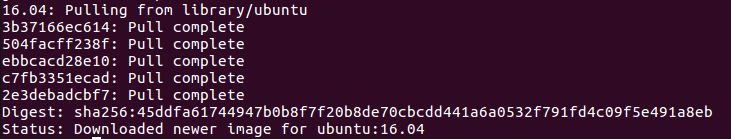


### 3.2 获取镜像

Docker Hub上有大量的高质量的镜像可以用,可以获取这些镜像来使用。其命令为docker pull  
输入如下命令:

docker pull ubuntu:16.04

可以看到输出如下信息:



### 3.3 列出镜像

要想列出本地的镜像,可以使用docker image ls命令。

docker image ls

输出信息如下:

1469241-20180914215101104-1020562534.png

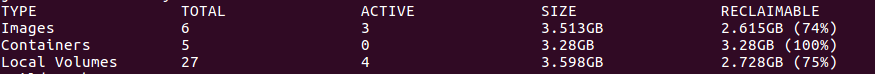
列表包含了仓库名,标签,镜像,ID,创建时间,以及所占用的空间。其中仓库名、标签在之前的基础概念章节已经介绍过了。镜像ID则是镜像的唯一标识,一个镜像可以对应多个标签。

另外一个需要注意的问题是,docker image ls列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker镜像是多层存储结构,并且可以继承、复用,因此不同镜像可能会因为使用相同的基础镜像,从而拥有共同的层。由于Docker使用Union FS,相同的层只需要保存一份即可,因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。

可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。

docker system df

输出结果如下:

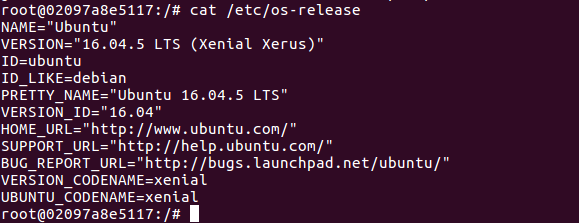


### 3.4 运行容器

有了镜像后,我们就能够以这个镜像为基础启动并运行一个容器。

docker run -it --rm ubuntu:16.04 /bin/bash

运行结果如下:

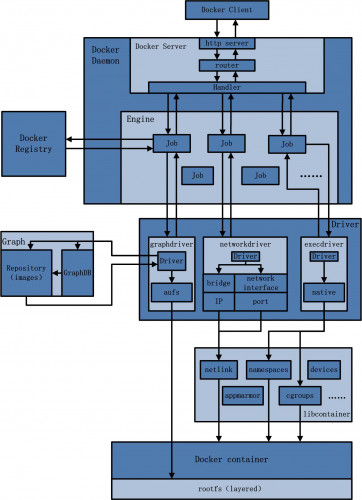


如上所示,就可以在容器中进行相关操作了.

### Docker总架构分解

Docker对使用者来讲是一个C/S模式的架构，而Docker的后端是一个非常松耦合的架构，模块各司其职，并有机组合，支撑Docker的运行。

在此，先附上Docker总架构：



从上图不难看出，用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。

而Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；而后Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。

Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过execdriver来完成。

而libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作。当执行完运行容器的命令后，一个实际的Docker容器就处于运行状态，该容器拥有独立的文件系统，独立并且安全的运行环境等。

**一、Docker架构内各模块的功能与实现分析**

接下来，我们将从Docker总架构图入手，抽离出架构内各个模块，并对各个模块进行更为细化的架构分析与功能阐述。主要的模块有：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer以及Docker container。

**1.1）Docker Client**

Docker Client是Docker架构中用户用来和Docker Daemon建立通信的客户端。用户使用的可执行文件为docker，通过docker命令行工具可以发起众多管理container的请求。

Docker Client可以通过以下三种方式和Docker Daemon建立通信：tcp://host:port，unix://path\_to\_socket和fd://socketfd。为了简单起见，本文一律使用第一种方式作为讲述两者通信的原型。与此同时，与Docker Daemon建立连接并传输请求的时候，Docker Client可以通过设置命令行flag参数的形式设置安全传输层协议(TLS)的有关参数，保证传输的安全性。

Docker Client发送容器管理请求后，由Docker Daemon接受并处理请求，当Docker Client接收到返回的请求相应并简单处理后，Docker Client一次完整的生命周期就结束了。当需要继续发送容器管理请求时，用户必须再次通过docker可执行文件创建Docker Client。

**1.2）Docker Daemon**

Docker Daemon是Docker架构中一个常驻在后台的系统进程，功能是：接受并处理Docker Client发送的请求，管理所有的 Docker 容器。。该守护进程在后台启动了一个Server，Server负责接受Docker Client发送的请求；接受请求后，Server通过路由与分发调度，找到相应的Handler来执行请求。

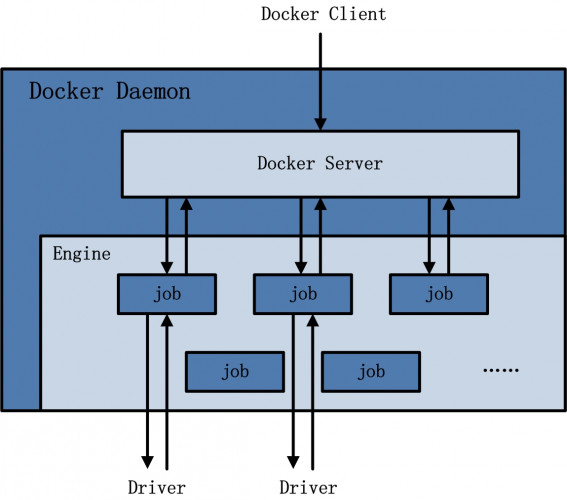
Docker Daemon启动所使用的可执行文件也为docker，与Docker Client启动所使用的可执行文件docker相同。在docker命令执行时，通过传入的参数来判别Docker Daemon与Docker Client。如 docker -d 代表 Docker Daemon 的启动，  
dockerps 则代表创建 Docker Client ，并发送 ps 请求。

Docker Daemon的架构，大致可以分为以下三部分：Docker Server、Engine和Job。Daemon架构如下图。

Docker：架构分解

**Docker Server**

Docker Server在Docker架构中是专门服务于Docker Client的server。该server的功能是：接受并调度分发Docker Client发送的请求。Docker Server的架构如下图。



在Docker的启动过程中，通过包gorilla/mux，创建了一个mux.Router，提供请求的路由功能。在Golang中，gorilla/mux是一个强大的URL路由器以及调度分发器。该mux.Router中添加了众多的路由项，每一个路由项由HTTP请求方法（PUT、POST、GET或DELETE）、URL、Handler三部分组成。

若Docker Client通过HTTP的形式访问Docker Daemon，创建完mux.Router之后，Docker将Server的监听地址以及mux.Router作为参数，创建一个httpSrv=http.Server{}，最终执行httpSrv.Serve()为请求服务。

在Server的服务过程中，Server在listener上接受Docker Client的访问请求，并创建一个全新的goroutine来服务该请求。在goroutine中，首先读取请求内容，然后做解析工作，接着找到相应的路由项，随后调用相应的Handler来处理该请求，最后Handler处理完请求之后回复该请求。

需要注意的是：Docker Server的运行在Docker的启动过程中，是靠一个名为”serveapi”的job的运行来完成的。原则上，Docker Server的运行是众多job中的一个，但是为了强调Docker Server的重要性以及为后续job服务的重要特性，将该”serveapi”的job单独抽离出来分析，理解为Docker Server。

**Engine**

Engine是Docker架构中的运行引擎，同时也Docker运行的核心模块。它扮演Docker container存储仓库的角色，并且通过执行job的方式来操纵管理这些容器。

在Engine数据结构的设计与实现过程中，有一个handler对象。该handler对象存储的都是关于众多特定job的handler处理访问。举例说明，Engine的handler对象中有一项为：{“create”: daemon.ContainerCreate,}，则说明当名为”create”的job在运行时，执行的是daemon.ContainerCreate的handler。

**Job**

一个Job可以认为是Docker架构中Engine内部最基本的工作执行单元。Docker可以做的每一项工作，都可以抽象为一个job。例如：在容器内部运行一个进程，这是一个job；创建一个新的容器，这是一个job，从Internet上下载一个文档，这是一个job；包括之前在Docker Server部分说过的，创建Server服务于HTTP的API，这也是一个job，等等。

Job的设计者，把Job设计得与Unix进程相仿。比如说：Job有一个名称，有参数，有环境变量，有标准的输入输出，有错误处理，有返回状态等。

**1.3）Docker Registry**

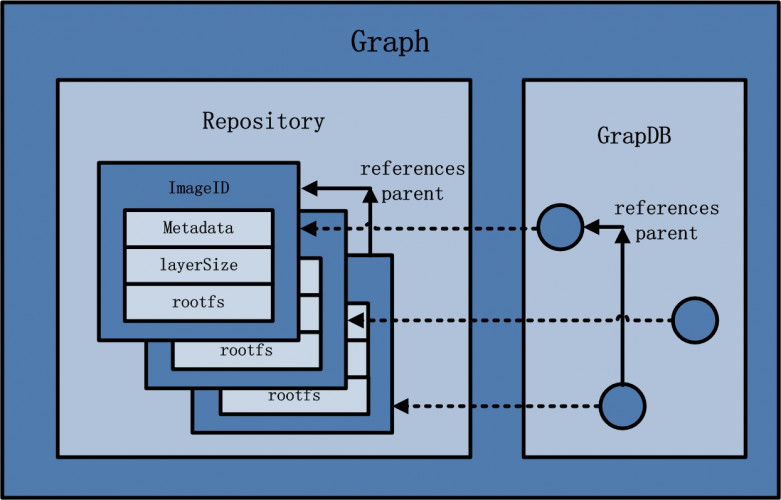
Docker Registry是一个存储容器镜像的仓库。而容器镜像是在容器被创建时，被加载用来初始化容器的文件架构与目录。

在Docker的运行过程中，Docker Daemon会与Docker Registry通信，并实现搜索镜像、下载镜像、上传镜像三个功能，这三个功能对应的job名称分别为”search”，”pull” 与 “push”。

其中，在Docker架构中，Docker可以使用公有的Docker Registry，即大家熟知的Docker Hub，如此一来，Docker获取容器镜像文件时，必须通过互联网访问Docker Hub；同时Docker也允许用户构建本地私有的Docker Registry，这样可以保证容器镜像的获取在内网完成。

**1.4）Graph**

Graph在Docker架构中扮演已下载容器镜像的保管者，以及已下载容器镜像之间关系的记录者。一方面，Graph存储着本地具有版本信息的文件系统镜像，另一方面也通过GraphDB记录着所有文件系统镜像彼此之间的关系。Graph的架构如下图。



其中，GraphDB是一个构建在SQLite之上的小型图数据库，实现了节点的命名以及节点之间关联关系的记录。它仅仅实现了大多数图数据库所拥有的一个小的子集，但是提供了简单的接口表示节点之间的关系。

同时在Graph的本地目录中，关于每一个的容器镜像，具体存储的信息有：该容器镜像的元数据，容器镜像的大小信息，以及该容器镜像所代表的具体rootfs。

Graph 在 Docker 架构中扮演的角色是容器镜像的保管者。不论是 Docker 下载的镜像，  
还是 Docker 构建的镜像，均由 Graph 统一化管理。由于 Docker 支持多种不同的镜像存储方  
式，如 aufs 、 devicemapper 、 Btrfs 等，故 Graph 对镜像的存储也会因以上种类而存在一些差  
异。对 Docker 而言，同一种类型的镜像被称为一个 repository ，如名称为 ubuntu 的镜像都  
同属一个 repository ;而同一个 repository 下的镜像则会因 tag 存在差异而不同，如 ubuntu  
这个 reposit。可下有 tag 为 12.04 的镜像，也有 tag 为 14.04 的镜像。

**1.5）Driver**

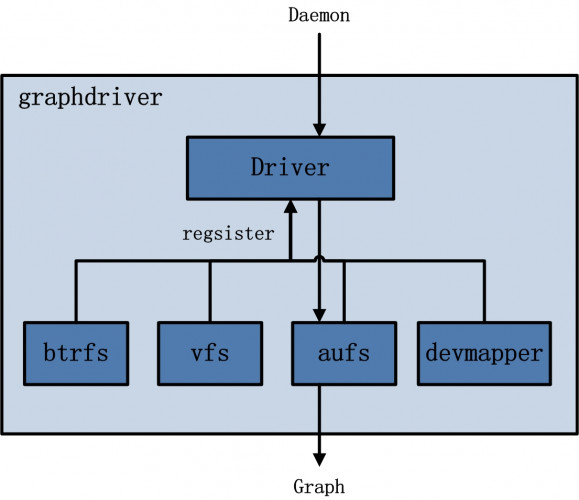
Driver是Docker架构中的驱动模块。通过Driver驱动，Docker可以实现对Docker容器执行环境的定制。由于Docker运行的生命周期中，并非用户所有的操作都是针对Docker容器的管理，另外还有关于Docker运行信息的获取，Graph的存储与记录等。因此，为了将Docker容器的管理从Docker Daemon内部业务逻辑中区分开来，设计了Driver层驱动来接管所有这部分请求。

在Docker Driver的实现中，可以分为以下三类驱动：graphdriver、networkdriver和execdriver。

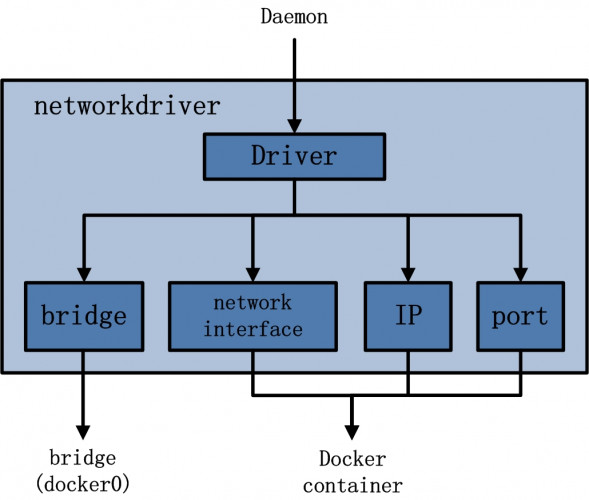
graphdriver主要用于完成容器镜像的管理，包括存储与获取。即当用户需要下载指定的容器镜像时，graphdriver将容器镜像存储在本地的指定目录；同时当用户需要使用指定的容器镜像来创建容器的rootfs时，graphdriver从本地镜像存储目录中获取指定的容器镜像。

在graphdriver的初始化过程之前，有4种文件系统或类文件系统在其内部注册，它们分别是aufs、btrfs、vfs和devmapper。而Docker在初始化之时，通过获取系统环境变量”DOCKER\_DRIVER”来提取所使用driver的指定类型。而之后所有的graph操作，都使用该driver来执行。

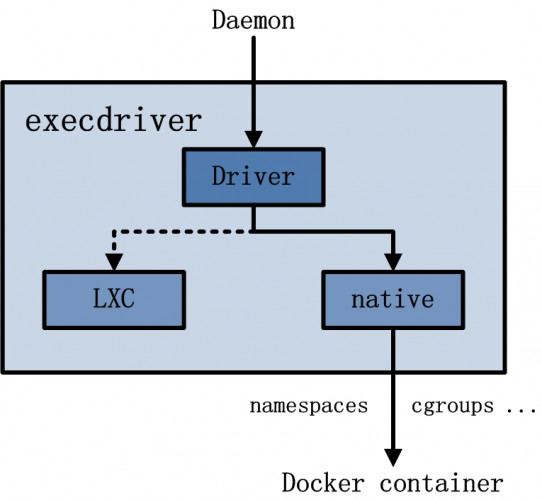
graphdriver的架构如图



networkdriver的用途是完成Docker容器网络环境的配置，其中包括Docker启动时为Docker环境创建网桥；Docker容器创建时为其创建专属虚拟网卡设备；以及为Docker容器分配IP、端口并与宿主机做端口映射，设置容器防火墙策略等。networkdriver的架构如下图。



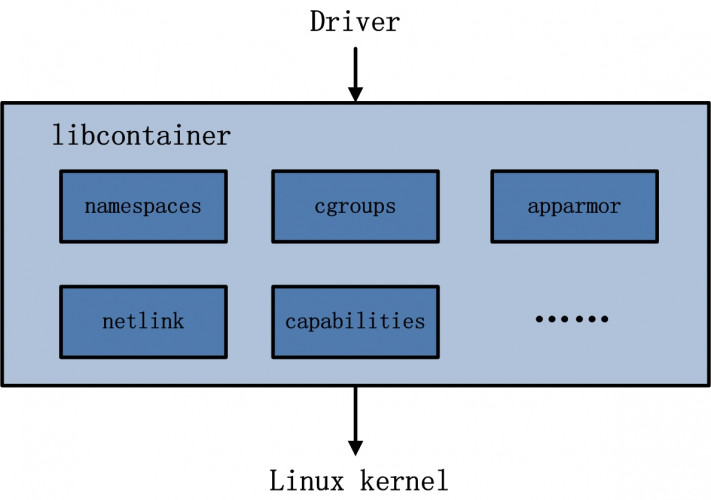
execdriver作为Docker容器的执行驱动，负责创建容器运行命名空间，负责容器资源使用的统计与限制，负责容器内部进程的真正运行等。在execdriver的实现过程中，原先可以使用LXC驱动调用LXC的接口，来操纵容器的配置以及生命周期，而现在execdriver默认使用native驱动，不依赖于LXC。具体体现在Daemon启动过程中加载的ExecDriverflag参数，该参数在配置文件已经被设为”native”。这可以认为是Docker在1.2版本上一个很大的改变，或者说Docker实现跨平台的一个先兆。execdriver架构如下图：



**1.6）libcontainer**

libcontainer是Docker架构中一个使用Go语言设计实现的库，设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的API。

正是由于libcontainer的存在，Docker可以直接调用libcontainer，而最终操纵容器的namespace、cgroups、apparmor、网络设备以及防火墙规则等。这一系列操作的完成都不需要依赖LXC或者其他包。libcontainer架构如下图：



另外，libcontainer提供了一整套标准的接口来满足上层对容器管理的需求。或者说，libcontainer屏蔽了Docker上层对容器的直接管理。又由于libcontainer使用Go这种跨平台的语言开发实现，且本身又可以被上层多种不同的编程语言访问，因此很难说，未来的Docker就一定会紧紧地和Linux捆绑在一起。而于此同时，Microsoft在其著名云计算平台Azure中，也添加了对Docker的支持，可见Docker的开放程度与业界的火热度。

暂不谈Docker，由于libcontainer的功能以及其本身与系统的松耦合特性，很有可能会在其他以容器为原型的平台出现，同时也很有可能催生出云计算领域全新的项目。

**1.10）Docker container**

Docker container（Docker容器）是Docker架构中服务交付的最终体现形式。Docker 通

过 DockerDaemon 的管理， libcontainer 的执行，最终创建 Docker 容器。

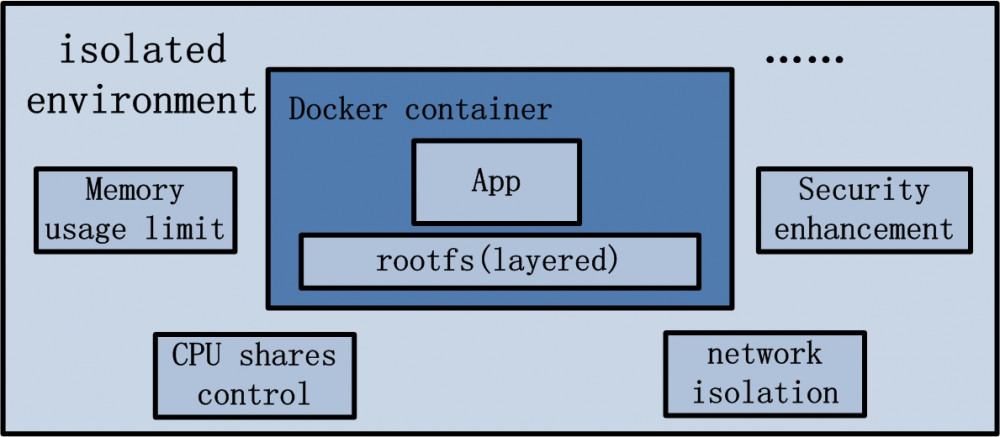
Docker 容器的从无到有，涉及 Docker 利用到的很多技术。总而言之，用户可以根据自

己的需求，通过 Docker C1ient 向 Docker Daemon 发送容器的创建与启动请求，请求中将携带

容器的配置信息，从而达到定制相应 Docker 容器的目的。用户对 Docker 容器的配置有以下

4 个基本方面:Docker按照用户的需求与指令，订制相应的Docker容器：

用户通过指定容器镜像，使得Docker容器可以自定义rootfs等文件系统； 用户通过指定计算资源的配额，使得Docker容器使用指定的计算资源； 用户通过配置网络及其安全策略，使得Docker容器拥有独立且安全的网络环境； 用户通过指定运行的命令，使得Docker容器执行指定的工作。



对于整个Docker的介绍先到这里，真正去理解Docker架构还需要多去实践Docker才行。

[Docker：容器介绍](https://www.cnblogs.com/zuxing/articles/8717153.html)

[Docker：架构分解](https://www.cnblogs.com/zuxing/articles/8717415.html)

[Docker：创建第一个容器](https://www.cnblogs.com/zuxing/articles/8745816.html)

[Dcoker：命令使用详解](https://www.cnblogs.com/zuxing/articles/8758600.html)